ProcessBase Standard Library Reference Manual

Version 1.0.4

October 1999

Ron Morrison†
Dharini Balasubramaniam†
Mark Greenwood¥
Graham Kirby†
Ken Mayes¥
Dave Munro*
Brian Warboys¥

†School of Mathematical and Computational Sciences,
University of St Andrews

¥Department of Computer Science,
University of Manchester

*Department of Computer Science,
University of Adelaide

Contents

1 Introduction

The ProcessBase library mechanism has two purposes. One is to allow separation of the ProcessBase language into a core part and a system dependent part. The core part of the language must be provided by all implementations, whereas the system dependent part may be implemented in different ways depending on the host system. The system dependent part includes persistence, IO, threads, strings, semaphores and mathematical functions. Libraries that implement these components on various platforms are supplied with the ProcessBase system, and are known as the Standard Libraries. They are described in this manual, the ProcessBase Standard Library Reference Manual.

There are two initial target implementations of ProcessBase: Solaris and Arena. Separate versions of the Standard Libraries will be implemented for each.

The second purpose of the library mechanism is to support extension of the ProcessBase architecture to achieve compliance for particular applications. The architecture may be extended at both the abstract machine (PBAM) level and at the ProcessBase language level. In the former case extension is achieved by defining new PBAM instructions, while in the latter case it is achieved by writing new ProcessBase source definitions in the core language. Thus a library may contain both new instructions and new source definitions—for example, most of the Standard Libraries contain both.

2 Using Libraries

Although libraries may contain both PBAM instructions and source code definitions, these elements are composed in different ways. The composition mechanism for PBAM instructions is static, in that a specific instantiation of the PBAM is built using a specific combination of libraries. This will normally comprise the versions of the Standard Libraries appropriate for the platform, together with any additional libraries designed for compliance to a particular application. Once built, the instruction set of that instantiation of PBAM is fixed.

Library source code definitions are composed on a per-compilation basis. For each compilation unit, the programmer indicates which libraries are required. This may be done either by passing additional parameters to the compiler, or by including a stylised comment at the beginning of the user code. An example of the former is:

pbc user-source.P maths-lib.path thread-lib.path

which instructs the ProcessBase compiler that the values defined in the Maths and Thread libraries should be in scope before the program *user-source.P* is compiled. Alternatively the libraries could be omitted from the compilation command, and the following comment included as the first line of *user-source.P*:

! Libraries: maths-lib.path thread-lib.path

It is envisaged that in future versions of ProcessBase the libraries will be pre-evaluated and the resulting values held in the persistent store. In this case some mechanism will be required to bind user programs to the library values.

In the meantime, library source code is held in files. A user program that uses libraries is defined to be equivalent to the result of inserting the source code of those libraries at the head of the user program.

3 The Standard Libraries

The contents of the Standard Libraries are now specified. Each library contains a collection of abstract machine instructions and ProcessBase source definitions. The source definitions use the ProcessBase down-call mechanism where appropriate.

3.1 Platform-Independent Libraries

3.1.1 The Maths Library

The Maths Standard Library supports various trigonometric and mathematical operations.

3.1.1.1 PBAM Code

The library contains the following PBAM instructions:

Instruction	Op-Code	Description
sin	200	Pop the floating point value *R* from the main stack; *R* is an angle in radians. Calculate the sine of *R*. **if** an arithmetic error occurs **then** **arithmetical exception** **else** Push the value of the sine of *R* onto the main stack.

Instruction	Op-Code	Description
cos	201	Pop the floating point value *R* from the main stack; *R* is an angle in radians. Calculate the cosine of *R*. **if** an arithmetic error occurs **then** **arithmetical exception** **else** Push the value of the cosine of *R* onto the main stack.

Instruction	Op-Code	Description
exp	202	Pop the floating point value *R* from the main stack. Calculate *e* raised to the power of *R*. **if** an arithmetic error occurs **then** **arithmetical exception** **else** Push the value of *e* raised to the power of *R* onto the main stack.

Instruction	Op-Code	Description
ln	203	Pop the floating point value *R* from the main stack. Calculate the natural logarithm of *R*. **if** an arithmetic error occurs **then** **arithmetical exception** **else** Push the value of the natural logarithm of *R* onto the main stack.

Instruction	Op-Code	Description
sqrt	204	Pop the floating point value *R* from the main stack. Calculate the square root of *R*. **if** an arithmetic error occurs **then** **arithmetical exception** **else** Push the value of the square root of *R* onto the main stack.

Instruction	Op-Code	Description
atan	205	Pop the floating point value *R* from the main stack; *R* is an angle in radians. Calculate the arctangent of *R*. **if** an arithmetic error occurs **then** **arithmetical exception** **else** Push the value of the arctangent of *R* onto the main stack.

Instruction	Op-Code	Description
truncate	206	Pop the floating point value *R* from the main stack. Calculate the integer part of *R*. **if** an arithmetic error occurs **then** **arithmetical exception** **else** Push the value of the integer part of *R* onto the main stack.

Instruction	Op-Code	Description
float	207	Replace the integer at the top of the main stack with the equivalent real.

Instruction	Op-Code	Description
abs	208	Replace the integer at the top of the main stack with its absolute value.

Instruction	Op-Code	Description
rabs	209	Replace the real at the top of the main stack with its absolute value.

3.1.1.2 Source Code

The library contains the following source definitions:

```
let sin <-      fun (x : real) -> real; downcall sin[](x)
let cos <-      fun (x : real) -> real; downcall cos[](x)
let exp <-      fun (x : real) -> real; downcall exp[](x)
let ln <-       fun (x : real) -> real; downcall ln[](x)
let arctan <-   fun (x : real) -> real; downcall atan[](x)
let sqrt <-     fun (x : real) -> real; downcall sqrt[](x)
let truncate <- fun (x : real) -> int;  downcall truncate[](x)
let float <-    fun (x : int) -> real;  downcall float[](x)
let abs <-      fun (x : int) -> int;   downcall abs[](x)
let rabs <-     fun (x : real) -> real; downcall rabs[](x)

let maxint <-  2147483647
let epsilon <- 5.4210108624275221e-20
let pi <-      3.141592654
let maxreal <- 1.79769313486231470e+308
```

3.1.2 The String Library

The String Standard Library supports various operations on strings. The library-dependent root object (see Section 3.2.4) contains pointers to the empty string literal and to a vector of single character strings.

3.1.2.1 PBAM Code

The library contains the following PBAM instructions.

Instruction	Op-Code	Description
code	210	**atomic** [Pop the integer *n* from the main stack. Push a pointer to a string of length 1, containing one character that is *ASCII code (abs (n rem 128))*, onto the pointer stack.]

Instruction	Op-Code	Description
decode	211	**atomic** [Pop the pointer to a string from the pointer stack. **if** the string is empty **then** **string exception** **else** Push the ASCII code for the first character of the string onto the main stack.]

Instruction	Op-Code	Description
length	212	**atomic** [Pop the pointer to a string from the pointer stack. Push the length of the string onto the main stack.]

3.1.2.2 Source Code

The library contains the following source definitions:

```
let code <-   fun (n : int) -> string; downcall code[](n)
let decode <- fun (s : string) -> int; downcall decode[](s)
let length <- fun (s : string) -> int; downcall length[](s)

let fformat <- fun (r : real; w, d : int) -> string
begin
    let zero <- decode( "0" )
    let ipart <- loc(w)
    let z <-
    begin
        let i <- loc(0)
        let y <- loc(if r < 0.0 then -r else r)
        while 'y >= 1.0 do { y := 'y / 10.0 ; i := 'i + 1 }
        i
    end
    if d < 0 or 'z > 'ipart then ""
    else begin
        if 'ipart = 0 do ipart := 1
        let round <- loc(1.0)
        let scal <- loc(1.0)
        for i <- 1 to d do round := 'round / 10.0
        for i <- 1 to 'z do scal := 'scal / 10.0
        let y <- loc((( if r < 0.0 then -r else r ) + 0.5 * 'round ) * 'scal)
        let result <- loc("")
        for i <- 1 to 'z do
        begin
            y := 'y * 10.0
            let k <- truncate( 'y )
            result := 'result ++ ( if k = 10 then "10" else code( k + zero ) )
            y := 'y - float( k )
        end
        let k <- truncate( 'y * 10.0 )
        if k = 10 do { result := 'result ++ "1" ; y := 0.0 ; z := 1 }
        result := 'result ++ "."
        for i <- 1 to d do
        begin
            y := 'y * 10.0
            let k <- truncate( 'y )
            result := 'result ++ code( k + zero )
            y := 'y - float( k )
        end
        if 'result( 1|1 ) = "." do
        begin
            result := "0" ++ 'result
            ipart := 'ipart - 1
        end
        result := ( if r < 0.0 then "-" else " " ) ++ 'result
        for i <- 1 to 'ipart - 'z do result := " " ++ 'result
        'result
    end
end

let eformat <- fun (r : real; w, d : int) -> string
if w < 0 or d < 0 then ""
else begin
    let g <- loc(1.0)
    for i <- 1 to w do g := 'g * 10.0
    let h <- 'g / 10.0
    let y <- if r < 0.0 then loc(-r) else loc(r)
    let scal <- loc(0)
    if 'y ~= 0.0 do
      if 'y >= 'g
        then while 'y >= 'g do { y := 'y / 10.0 ; scal := 'scal + 1 }
        else while 'y < h do { y := 'y * 10.0 ; scal := 'scal - 1 }
    let st1 <- fformat( if r > 0.0 then 'y else - 'y, w, d )
```

```
    let st2 <-
    begin
        if 'scal = 0 then "e+00" else
        begin
            let scal1 <- abs( 'scal )
            let sign <- if 'scal < 0 then { scal := - 'scal ; true } else false
            let result <- loc("") ; let zero <- decode( "0" )
            while 'scal > 0 do
            begin
                result := code( 'scal rem 10 + zero ) ++ 'result
                scal := 'scal div 10
            end

            "e" ++ ( if sign then "-" else "+" ) ++
            ( if scal1 < 10 then "0" else "" ) ++ 'result
        end
    end

    st1 ++ st2
end

let gformat <- fun (r : real) -> string
if r = 0.0 then "0." else
begin
    let absR <- if r < 0.0 then -r else r
    let fpart <- if absR < 0.1 or absR > 1e5 then 0
                 else if absR < 1.0 then 6
                 else if absR < 10.0 then 5
                 else if absR < 100.0 then 4
                 else if absR < 1000.0 then 3
                 else if absR < 10000.0 then 2
                 else 1
    let ans <- loc("")
    let ePart <- loc("")
    if fpart = 0 then
    begin
        ans := eformat( r,1,5 )
        let lnth <- length( 'ans )
        ePart := 'ans( lnth - 3|4 )
        ans := 'ans( 1|lnth - 4 )
    end
    else ans := fformat( r, 7 - fpart, fpart )
    let pos <- loc(length( 'ans ))
    while 'ans( 'pos|1 ) = "0" do pos := 'pos - 1

    'ans( 1|'pos  ) ++ 'ePart
end

let iformat <- fun (n : int) -> string
begin
    let x <- loc(n)
    let sign <- if 'x < 0 then { x := - 'x ; "-" } else ""
    let result <- loc("")
    result := code( 48 + 'x rem 10 ) ++ 'result
    x := 'x div 10
    while 'x > 0 do
    begin
        result := code( 48 + 'x rem 10 ) ++ 'result
        x := 'x div 10
    end

    sign ++ 'result
end

let letter <- fun (s : string) -> bool
    length (s) = 1 and s >= "A" and s <= "Z" or s >= "a" and s <= "z"

let digit <- fun (s : string) -> bool
    length (s) = 1 and s >= "0" and s <= "9"
```

3.1.3 The Persistence Library

The Persistence Standard Library supports orthogonal persistence.

3.1.3.1 Source Code

The library contains the following source definition:

```
let PS ← fun() → any; downcall userRoot[]()
```

3.1.4 The Exception Library

The Exception Standard Library defines the types used in manipulating exceptions.

3.1.4.1 Source Code

The library contains the following source definition:

```
type Exception is view [name, description : string]
```

3.1.5 The Safe Op-code Library

The Safe Op-code Standard Library defines the op-codes that may be invoked from ProcessBase by down-calls, with a guarantee of type-safety.

3.1.5.1 Source Code

The library contains the following source definitions:

```
! Core interpreter instructions.
let userRoot ←     opcode 22[5] () → any
let concatenate ← opcode 84[]  (string, string) → string
let subString ←   opcode 85[]  (int, int, string) → string
let eqR ←         opcode 111[] (real, real) → bool
let eqS ←         opcode 112[] (string, string) → bool
let eqAny ←       opcode 115[] (any, any) → bool
let neqR ←        opcode 117[] (real, real) → bool
let neqS ←        opcode 118[] (string, string) → bool
let ltI ←         opcode 122[] (int, int) → bool
let ltR ←         opcode 123[] (real, real) → bool
let ltS ←         opcode 124[] (string, string) → bool
let leI ←         opcode 126[] (int, int) → bool
let leR ←         opcode 127[] (real, real) → bool
let leS ←         opcode 128[] (string, string) → bool
let gtI ←         opcode 130[] (int, int) → bool
let gtR ←         opcode 131[] (real, real) → bool
let gtS ←         opcode 132[] (string, string) → bool
let geI ←         opcode 134[] (int, int) → bool
let geR ←         opcode 135[] (real, real) → bool
let geS ←         opcode 136[] (string, string) → bool
let plus ←        opcode 140[] (int, int) → int
let times ←       opcode 141[] (int, int) → int
let minus ←       opcode 142[] (int, int) → int
let div ←         opcode 143[] (int, int) → int
```

```
let neg ←          opcode 144[] (int) → int
let rem ←          opcode 145[] (int, int) → int
let not ←          opcode 146[] (bool) → bool
let fPlus ←        opcode 150[] (real, real) → real
let fTimes ←       opcode 151[] (real, real) → real
let fMinus ←       opcode 152[] (real, real) → real
let fDivide ←      opcode 153[] (real, real) → real
let fNeg ←         opcode 154[] (real) → real
let hashCode ←     opcode 172[] (any) → int

! Standard library instructions.
let sin ←                   opcode 200[]  (real) → real
let cos ←                   opcode 201[]  (real) → real
let exp ←                   opcode 202[]  (real) → real
let ln ←                    opcode 203[]  (real) → real
let sqrt ←                  opcode 204[]  (real) → real
let atan ←                  opcode 205[]  (real) → real
let truncate ←              opcode 206[]  (real) → int
let float ←                 opcode 207[]  (int) → real
let abs ←                   opcode 208[]  (int) → int
let rabs ←                  opcode 209[]  (real) → real
let code ←                  opcode 210[]  (int) → string
let decode ←                opcode 211[]  (string) → int
let length ←                opcode 212[]  (string) → int

let readOp ←                opcode 214[0] (int) → string
let peekOp ←                opcode 214[1] (int) → string
let readIntOp ←             opcode 214[2] (int) → int
let readBoolOp ←            opcode 214[3] (int) → bool
let readStringOp ←          opcode 214[4] (int) → string
let readRealOp ←            opcode 214[5] (int) → real
let readByteOp ←            opcode 214[6] (int) → int
let readLineOp ←            opcode 214[7] (int) → string
let eofOp ←                 opcode 214[8] (int) → bool

let writeIntOp ←            opcode 215[0] (int, int, int)
let writeBoolOp ←           opcode 215[1] (int, bool, int)
let writeStringOp ←         opcode 215[2] (int, string, int)
let writeRealOp ←           opcode 215[3] (int, real, int)
let writeByteOp ←           opcode 215[4] (int, int, int)

let openOp ←                opcode 216[]  (string, int) → int
let createOp ←              opcode 217[]  (string) → int
let closeOp ←               opcode 218[]  (int) → int

let newThreadOp ←           opcode 221[0] (fun ()) → int
let getThreadIdOp ←         opcode 221[1] () → int
let killThreadOp ←          opcode 221[2] (int)
let resumeThreadOp ←        opcode 221[3] (int)
let suspendThreadOp ←       opcode 221[4] (int)
let getThreadStatusOp ←     opcode 221[5] (int) → int

let semaphoreCreateOp ←     opcode 222[0] (int) → int
let semaphoreWaitOp ←       opcode 222[1] (int)
let semaphoreSignalOp ←     opcode 222[2] (int)
```

```
let interruptOp ←          opcode 223[int] (string)
```

3.1.6 The Unsafe Opcode Library

The Unsafe Opcode Standard Library defines the op-codes that may be invoked from ProcessBase by down-calls, with a risk of compromising type-safety if used inconsistently.

3.1.6.1 Source Code

The library contains the following source definitions:

```
! Core interpreter instructions.
let fJump ←       opcode 0[int]
let bJump ←       opcode 1[int]
let jumpF ←       opcode 2[int]
let jumpFF ←      opcode 4[int]
let jumpTT ←      opcode 5[int]
let forTest ←     opcode 6[int]
let forStep ←     opcode 7[int]

let wAssign ←     opcode 10[]
let dwAssign ←    opcode 11[]
let pAssign ←     opcode 12[]
let dpAssign ←    opcode 13[]
let wVassign ←    opcode 14[]
let dwVassign ←   opcode 15[]
let pVassign ←    opcode 16[]
let dpVassign ←   opcode 17[]

let wRoot ←       opcode 20[int]
let dwRoot ←      opcode 21[int]
let pRoot ←       opcode 22[int]
let dpRoot ←      opcode 23[int]
let wLocal ←      opcode 24[int]
let dwLocal ←     opcode 25[int]
let pLocal ←      opcode 26[int]
let dpLocal ←     opcode 27[int]
let wDeref ←      opcode 28[]
let dwDeref ←     opcode 29[]
let pDeref ←      opcode 30[]
let dpDeref ←     opcode 31[]
let wSubVector ←  opcode 32[]
let dwSubVector ← opcode 33[]
let pSubVector ←  opcode 34[]
let dpSubVector ← opcode 35[]
let wSubView ←    opcode 36[int]
let dwSubView ←   opcode 37[int]
let pSubView ←    opcode 38[int]
let dpSubView ←   opcode 39[int]
let wEnv ←        opcode 40[int]
let dwEnv ←       opcode 41[int]
let pEnv ←        opcode 42[int]
let dpEnv ←       opcode 43[int]
let wProject ←    opcode 44[]
```

```
let dwProject ←   opcode 45[]
let pProject ←    opcode 46[]
let dpProject ←   opcode 47[]
let loadAnyType ← opcode 48[]
let wLiteral ←    opcode 50[int, int]
let dwLiteral ←   opcode 51[int, int]
let pLiteral ←    opcode 52[int, int]
let dpLiteral ←   opcode 53[int, int]
let lLInt ←       opcode 54[int]
let lLChar ←      opcode 55[int]

let wRetract ←    opcode 60[int, int]
let dwRetract ←   opcode 61[int, int]
let pRetract ←    opcode 62[int, int]
let dpRetract ←   opcode 63[int, int]
let retract ←     opcode 64[int, int]

let markStack ←   opcode 70[]
let apply ←       opcode 71[int, int]
let wReturn ←     opcode 74[]
let dwReturn ←    opcode 75[]
let pReturn ←     opcode 76[]
let dpReturn ←    opcode 77[]
let return ←      opcode 78[]

let wMakeLoc ←    opcode 80[]
let dwMakeLoc ←   opcode 81[]
let pMakeLoc ←    opcode 82[]
let dpMakeLoc ←   opcode 83[]

let wMakeVector ←   opcode 88[int]
let dwMakeVector ←  opcode 89[int]
let pMakeVector ←   opcode 90[int]
let dpMakeVector ←  opcode 91[int]
let wMakeEvec ←     opcode 92[]
let dwMakeEvec ←    opcode 93[]
let pMakeEvec ←     opcode 94[]
let dpMakeEvec ←    opcode 95[]

let makeView ←    opcode 96[int, int]
let formClosure ← opcode 98[int, int]

let wMakeAny ←    opcode 100[int]
let dwMakeAny ←   opcode 101[int]
let pMakeAny ←    opcode 102[int]
let dpMakeAny ←   opcode 103[int]

let eqIB ←        opcode 110[]
let eqP ←         opcode 113[]
let eqPr ←        opcode 114[]
let neqIB ←       opcode 116[]
let neqP ←        opcode 119[]
let neqPr ←       opcode 120[]

let enterEHandled ←  opcode 160[int]
```

```
let exitEHandled ←   opcode 161[]
let invokeEHandler ← opcode 162[]
let enterIHandled ←  opcode 163[int]
let exitIHandled ←   opcode 164[]

let lwb ←          opcode 170[]
let upb ←          opcode 171[]

! Standard library instructions.
let read ←         opcode 214[int]
let write ←        opcode 215[int]
let threadOp ←     opcode 221[int]
let semaphoreOp ← opcode 222[int]
```

3.2 Solaris Libraries

3.2.1 The Solaris IO Library

The Solaris IO Standard Library supports various file operations on the Solaris platform.

3.2.1.1 File Data Structure

For each file, created or opened, a File Control Block (FCB) is allocated from the heap and added to a list of FCBs pointed to by word 2 of the library-dependent root object (see Section 3.2.4). The FCB is indexed by the integer file descriptor and has the following format:

word 0,1	object header and size
word 2	pointer to next FCB or **nil** for end of list
word 3	flags and file descriptor
word 4	cp: current position in the buffer
word 5	number of bytes in the buffer
word 6..n	buffer
word n+1	unused (reserved for hash code but never used since FCB cannot be manipulated directly by user programs)

The following operations are permitted on a file:

- *equals, not equals,*
- *read, peek, readInt, readBool, readString, readReal, readByte, readLine, eof,*
- *writeInt, writeBool, writeString, writeReal, writeByte.*

3.2.1.2 PBAM Code

The library contains the following PBAM instructions.

Instruction	Op-Code	Description

readOp (n:short)	214	**atomic** [*n* indicates which read operation to perform. Pop the file descriptor from the top of the main stack. Read a value from the file and push it onto the appropriate stack.]

The read operations are:

Operation	Code	Description
read	0	read the next character in the file.
peek	1	look at the next character and return it without reading it.
readInt	2	skip tab, space and newline characters and read an integer literal.
readBool	3	skip tab, space and newline characters and read a boolean literal.
readString	4	skip tab, space and newline characters and read a string literal.
readReal	5	skip tab, space and newline characters and read a real literal.
readByte	6	read one 8 bit byte as an integer.
readLine	7	read from the current position up to a newline symbol. Give the result as a string without the newline symbol.
eof	8	test for end of file.

Instruction	Op-Code	Description
writeOp (n:short)	215	**atomic** [*n* indicates which operation to perform. Pop the field width from the main stack. Pop the value to be written out from the appropriate stack. Pop the file descriptor from the main stack. Write the value to the file.]

The write operations are:

Operation	Code	Description
writeInt	0	write an integer
writeBool	1	write a boolean
writeString	2	write a string
writeReal	3	write a real
writeByte	4	write an 8 bit byte.

Instruction	Op-Code	Description
openOp	216	**atomic** [Pop a pointer to a string containing the name of the file from the pointer stack. Pop an integer access mode (0 - read, 1 - write) from the main stack. Create a new file control block (FCB) object and add it to the list of files in the library-dependent root object. Open the named file. **if** the specified file does not exist **then** raise **I/O exception**. **else** Push the integer file descriptor onto the main stack.]

Instruction	Op-Code	Description
createOp	217	**atomic** [Pop a pointer to a string containing the name of the file from the pointer stack. Create a new file control block (FCB) object and add it to the list of files in the library-dependent root object. Create the named file. **if** the creation fails **then** raise **I/O exception**. **else** Push the integer file descriptor onto the main stack.]

Instruction	Op-Code	Description
closeOp	218	**atomic** [Pop an integer file descriptor from the main stack. Close the file. Remove the corresponding file control block (FCB) from the list of files in the library-dependent root object.]

3.2.1.3 Source Code

The library contains the following source definitions:

```
let read <-       fun (f : int) -> string; downcall readOp[](f)
let peek <-       fun (f : int) -> string; downcall peekOp[](f)
let readInt <-    fun (f : int) -> int;    downcall readIntOp[](f)
let readBool <-   fun (f : int) -> bool;   downcall readBoolOp[](f)
let readString <- fun (f : int) -> string; downcall readStringOp[](f)
let readReal <-   fun (f : int) -> real;   downcall readRealOp[](f)
let readByte <-   fun (f : int) -> int;    downcall readByteOp[](f)
let readLine <-   fun (f : int) -> string; downcall readLineOp[](f)
let eof <-        fun (f : int) -> bool;   downcall eofOp[](f)
```

```
let writeInt <-    fun (f: int; x: int; n: int);    downcall writeIntOp[](f, x, n)
let writeBool <-   fun (f: int; x: bool; n: int);   downcall writeBoolOp[](f, x, n)
let writeString <- fun (f: int; x: string; n: int); downcall writeStringOp(f, x, n)
let writeReal <-   fun (f: int; x: real; n: int);   downcall writeRealOp(f, x, n)
let writeByte <-   fun (f: int; x: int; n: int);    downcall writeByteOp(f, x, n)

let open <-   fun (s : string; m : int) -> int; downcall openOp[](s, m)
let create <- fun (s : string) -> int;          downcall createOp[](s)
let close <-  fun (f : int);                    downcall closeOp[](f)

let r_w <- loc (14)
let s_w <- loc (2)
let i_w <- loc (12)
let si <-  loc (open ("STDIN:", 0))
let so <-  loc (create ("STDOUT:"))
```

3.2.1.4 Exceptions

name	description	circumstances
"io"	description of the attempted operation and parameters	if an I/O error occurs

3.2.2 The Solaris Thread Library

The Solaris Thread Standard Library supports a subset of Solaris thread functions. Future libraries may include sets of functions to encompass POSIX functionality and Solaris thread package functionality.

When the PBAM is initialised, one thread is created automatically to execute the main program. Further threads may be created explicitly using the appropriate library procedures. The PBAM shuts down when there are no remaining runnable threads.

3.2.2.1 Thread Data Structure

For each thread created, a Thread Control Block (TCB) is allocated from the heap and added to a list of TCBs pointed to by word 3 of the library-dependent root object (see Section 3.2.4). The TCB holds the values for the PBAM registers appropriate to that thread, and has the following format:

word 0,1	object header and size
word 2	pointer to next TCB or **nil** for end of list
word 3	pointer to stack object
word 4	environment pointer
word 5	exception handler stack pointer
word 6	interrupt handler stack pointer
word 7	code pointer
word 8	pointer stack frame base
word 9	pointer stack top
word 10	main stack frame base
word 11	main stack top
word 12	ID of PBAM thread
word 13	ID of corresponding operating system thread
word 14	status flags

word 15	unused (reserved for hash code but never used since thread cannot be manipulated directly by user programs)

3.2.2.2 PBAM Code

The library contains the following PBAM instructions.

Instruction	Op-Code	Description
threadOp (n:short)	221	*n* indicates which thread operation to perform.

The thread operations are:

Operation	Code	Description
newThread	0	**atomic** [Pop the environment pointer for the thread's procedure from the pointer stack. Pop the code vector for the thread's procedure from the pointer stack. Create a new thread control block (TCB) object. Create a new pointer stack and main stack for the thread. Copy the pointer stack base pointer, pointer stack top pointer, main stack base pointer and main stack top pointer to words 8..11 of the TCB respectively. Copy the code vector pointer to word 7 of the TCB. Copy the environment pointer to word 4 of the TCB. Generate a PBAM thread identifier for the new thread and copy it to word 12 of the TCB. Create a new Solaris thread to perform the following: Push **nil** for IRP onto the new main stack. Allocate a C *jmp_buf* array and push its address, IRC, onto the new main stack. Push a MSCW with **nil** dynamic link onto the new main stack. Record the current C execution context in the *jmp_buf* array. Start executing the new procedure. Copy the identifier of the Solaris thread to word 13 of the TCB. Set the status flags in word 14 of the TCB to *runnable*. Add the TCB to the list of threads in the library-dependent root object.] Push the PBAM thread identifier onto the main stack.

Operation	Code	Description
getThreadId	1	Push the PBAM identifier for the current thread, in word 12 of the corresponding TCB, onto the main stack.

Operation	Code	Description
killThread	2	**atomic** [Pop the PBAM identifier of a thread from the main stack. Scan the list of threads in the library-dependent root object for a TCB containing the specified PBAM thread identifier in word 12. **if** the TCB is found **do** Remove the TCB from the list. **if** the specified PBAM thread is the current thread **then** Exit from the current Solaris thread. **else** Send a kill signal to the Solaris thread identified by word 13 of the TCB.]

Operation	Code	Description
resumeThread	3	**atomic** [Pop the PBAM identifier of a thread from the main stack. Scan the list of threads in the library-dependent root object for a TCB containing the specified PBAM thread identifier in word 12. **if** the TCB is found **do** Set the status flags in word 14 of the TCB to *runnable*. Resume the Solaris thread identified by word 13 of the TCB.]

Operation	Code	Description
suspendThread	4	**atomic** [Pop the PBAM identifier of a thread from the main stack. Scan the list of threads in the library-dependent root object for a TCB containing the specified PBAM thread identifier in word 12. **if** the TCB is found **do** Set the status flags in word 14 of the TCB to *suspended.* Suspend the Solaris thread identified by word 13 of the TCB.]

Operation	Code	Description
getThreadStatus	5	**atomic** [Pop the PBAM identifier of a thread from the main stack. Scan the list of threads in the library-dependent root object for a TCB containing the specified PBAM thread identifier in word 12. **if** the TCB is found **then** Read the status of the PBAM thread from the flags in word 14 of the TCB. **if** the PBAM thread is runnable **then** Push 1 (*runnable*) onto the main stack. **else** Push 2 (*suspended*) onto the main stack. **else** Push 0 (*does not exist*) onto the main stack.]

3.2.2.3 Source Code

The library contains the following source definitions:

```
let start <-      fun (fn : fun()) -> int;  downcall newThreadOp[](fn)
let getCurrent <- fun () -> int;            downcall getThreadIdOp[]()
let kill <-       fun (x : int);            downcall killThreadOp[](x)
let resume <-     fun (x : int);            downcall resumeThreadOp[](x)
let suspend <-    fun (x : int);            downcall suspendThreadOp[](x)
let getStatus <-  fun (x: int) -> int;      downcall getThreadStatusOp[](x)
```

3.2.3 The Solaris Semaphore Library

The Solaris Semaphore Standard Library supports semaphores on the Solaris platform.

3.2.3.1 PBAM Code

The library contains the following PBAM instruction.

Operation	Code	Description
semaphoreOp (n:short)	222	**if** *n = 0* **then** Pop an initial integer value *i* for a semaphore from the top of the main stack. Create a Solaris semaphore *struct*, initialised with *i*. Push a C-pointer to the semaphore *struct* onto the main stack. **else** Pop a C-pointer to a Solaris semaphore *struct* from the main stack. **if** *n = 1* **then** Perform a *wait* operation on the semaphore. **else** Perform a *signal* operation on the semaphore.

3.2.3.2 Source Code

The library contains the following source definitions:

```
type semaphore is view [wait : fun(); signal : fun()]

let newSemaphore <- fun (count : int) -> semaphore
begin
    let semId <- downcall semaphoreCreateOp[](count)

    view (wait <-   fun(); downcall semaphoreWaitOp[](semId),
          signal <- fun(); downcall semaphoreSignalOp[](semId) )
end
```

3.2.4 The Solaris Interrupt Library

The Solaris Interrupt Standard Library defines the interrupts that may be raised by the Solaris PBAM implementation, and an instruction that may be used to control interrupts.

3.2.4.1 PBAM Code

The library contains the following PBAM instruction.

Operation	Code	Description
interruptOp (n:short)	223	**atomic** [Pop a pointer to a string parameter from the pointer stack. *n* specifies the interrupt with which the parameter is to be associated. Record the interrupt identifier and parameter where they can be accessed by the code implementing the interrupt.]

3.2.4.2 Source Code

The library contains the following source definitions:

```
! Interrupt raised to running threads to indicate Solaris hangup signal.
```

```
! Initially enabled; can be switched on or off using interruptOp with
! parameters "enable" and "disable".
let system_hangup ← interrupt 0()

! Interrupt raised to running threads to indicate Solaris interrupt signal.
! Initially enabled; can be switched on or off using interruptOp with
! parameters "enable" and "disable".
let system_interrupt ← interrupt 1()

! Interrupt raised to running threads to indicate Solaris quit signal.
! Initially enabled; can be switched on or off using interruptOp with
! parameters "enable" and "disable".
let system_quit ← interrupt 2()

! Interrupt raised to running threads giving number of ticks since system initialisation.
! Initially enabled; can be switched on or off using interruptOp with
! parameters "enable" and "disable".
let timer ← interrupt 3(int)
```

4 The Library-Dependent Root Object

Certain instructions in the Standard Libraries operate on persistent data structures, such as the file and thread instructions which manipulate a File Control Block list and a Thread Control Block list respectively. In order to maintain the separation of core and library components, these data structures are not allocated space in the root object directly. Instead they are reachable from the *library-dependent root object*, which is itself pointed to by word 10 of the root object.

The library-dependent root object has the following format:

word 0,1	object header and size
word 2	pointer to first element of File Control Block List or **nil** if empty
word 3	pointer to first element of Thread Control Block List or **nil** if empty
word 4	pointer to the empty string literal ""
word 5	pointer to a vector of all 128 single character strings
word 6	unused (reserved for hash code but never used since object cannot be manipulated directly by user programs)

Index